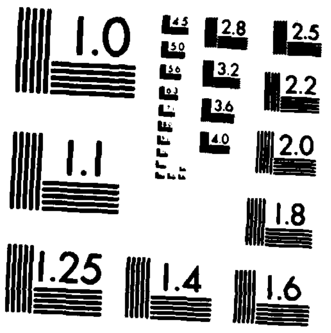
AD-A129 696 THE CUE (CONSISTENT UNDERLYING ENVIRONMENT) PROJECT(U) 1/1
UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY
INFORMATION SCIENCES INST T KACZMAREK ET AL. MAY 83
UNCLASSIFIED ISI/RS-83-1 MDA903-81-C-0335 F/G 9/2 NL

END
DATE
FILMED
7 83
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(12)

ISI Reprint Series
ISI/RS-83-1
May 1983

University of Southern California

Thomas Kaczmarek
William Mark
David Wilczynski

The CUE Project

INFORMATION SCIENCES INSTITUTE
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RS-83-1	2. GOVT ACCESSION NO. AD-A129696	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The CUE Project		5. TYPE OF REPORT & PERIOD COVERED Research Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Thomas Kaczmarek, William Mark, and David Wilczynski		8. CONTRACT OR GRANT NUMBER(s) MDA903 81 C 0335
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ....
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1983
		13. NUMBER OF PAGES 16
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

....

18. SUPPLEMENTARY NOTES

This report is a reprint of an article that appears in the proceedings of the SOFTFAIR '83 Conference, held in Arlington, Virginia, in July 1983. The SOFTFAIR Conference is jointly sponsored by the IEEE Computer Society, ACM, and the National Bureau of Standards.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer software design/development, artificial intelligence, programming environments, program analysis, software methodologies, the user interface

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The goal of the CUE Project is to produce a system environment in which the data and functions of different interactive services can be combined to perform a single computation task. Construction of this environment will require research on data representation, program functionalization, user interface design, and selection of appropriate levels for inter-machine communication. CUE is designed to take advantage of the knowledge-based system technology being developed in the Consul project. The CUE project will result in a running system demonstrating integrated services and exhibiting automatic interaction with the Consul system when required.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ISI Reprint Series
ISI/RS-83-1
May 1983

University of Southern California

Thomas Kaczmarek
William Mark
David Wilczynski

The CUE Project

INFORMATION SCIENCES INSTITUTE
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

ISI Reprint Series

CONTENTS

1. Introduction

The goal of the CUE project is to provide an extensible environment for building and using integrated interactive computer services. In CUE, there are no boundaries between say, the electronic mail service, and the automatic calendar service. Consider the following task:

sending a message to everyone in my 3:30 meeting to tell them I can't make it

Accomplishing this task requires a common understanding of the services involved: essentially, one must know how to find a particular meeting in the calendar system framework, access its attendees, and convert them to valid addressees in the message system framework. Unless the basis for such understanding has been designed in from the very beginning, automatic inter-service interaction at this level is impossible. In current systems, almost all of which consist of totally separate service programs or "subsystems", the necessary inter-service understanding is completely outside of the design framework. If a user wants to perform a task like the one above, he must interact with each service program separately, doing his own data conversions (usually via a text editor). In short, given current software environments, the problems of such a priori program planning and implementation to allow for inter-service interaction are intractable. Tasks like the one above cannot be performed without a great deal of painful user involvement.

Automatic inter-service interaction requires an environment in which individual services can make assumptions about the properties and behavior of the other services in the system. These assumptions must relate to data structures, functional capability, and state information. The objective of the CUE project is to produce such a consistent underlying environment (CUE) in which users can interact with the system without regard to service boundaries.

2. The CUE Concept

The basic idea of CUE is to influence the building of services so that they can later be used in an integrated way. The key is to have a semantic model of how each program element (function or data structure) fits in with the other program elements in the system. That is, when a "send message" function is defined, CUE must have a model of how the elements of a message (e.g., the "user" in the addressee field) are already being used in the system. It must also know that "sending" involves transmitting information from one place to another, that that information must be made accessible at the destination, and so on. CUE can then use this knowledge to ensure that each new program element fits in appropriately with, and can be used as a part of, overall system functionality. CUE must also act as an execution environment which supports the sharing (or coercion) of data structures and uniformity of function invocation necessary for inter-service interaction.

The purpose of such a service building methodology is to allow users to have a consistent view of the system. CUE will provide an command-oriented user interface which reflects the system's internal integration. The user will be able to invoke commands on structures defined in different services, and to sequence commands from different services to achieve a single task. As an added bonus (due to the knowledge-based service building methodology), the user will be able to automatically call upon more powerful knowledge-based user interface capabilities when necessary.

This report is a reprint of an article that appears in the proceedings of the SOFTFAIR '83 Conference, held in Arlington, Virginia, in July 1983. The SOFTFAIR Conference is jointly sponsored by the IEEE Computer Society, ACM, and the National Bureau of Standards.

To take full advantage of CUE's underlying consistency, the user interface must facilitate two activities: (1) sharing data from various services and (2) combining functions from various services. All interaction with CUE will be through a multi-windowed full-screen editor. Such an interface is ideal for encouraging data sharing because it allows the user (through movement of a graphic cursor) to select any displayed information and to insert a copy of it at any position on the display. Functional combination will be encouraged by allowing arbitrary functions to be applied to any selected region of a window and by supporting a functional style of computing in the command language.

2.1 Research Issues

The CUE project must address the following issues:

[1] specifying a common data definition substrate;

[2] representing the relationships among structures and functions of different services;

[3] defining the user interface;

[4] achieving the correct level of modularity (grain size) for service functions;

[5] finding the correct level of interaction with a knowledge-based user interface.

Although there are currently no adequate solutions to these problems, we believe that there are fruitful approaches for all of them, based in part on complementary research at ISI and elsewhere.

2.2 Approaches to the Solution

The problem of common data definition is well known in computer science: several kinds of solutions have been or are being investigated (e.g., [Yntema 73], [IPAD 80]). One of these techniques or a combination of them must be adapted by CUE to its service definition formalism. Work on abstract data types ([Dahl 72]) will have a very strong influence on CUE. This technique is appealing because of its compatibility with the requirements of knowledge-based modelling and because it provides a well-defined and consistent view of data structures.

A few current systems (e.g., UNIX) contain primitive mechanisms for combining the functional elements and data structures of different interactive services. However, there is never a semantic model of the relationships between different service entities. Thus, there is no representation of what, say a message service and a calendar service, have as a basis for commonality. Existing systems can therefore never go very far toward ensuring integration: they must rely on each service builder to think of the necessary interrelationships between his service and others, and to design in enough flexibility to handle any interactions that might occur.

CUE will avoid this total reliance on the service builder by providing a semantic model that links the functions and data structures of one service to those of others. It will provide an acquisition mechanism that associates the program elements of new services with the constructs of this model. Research on knowledge representation and acquisition in the Consul project provides already well-developed techniques for modelling the characteristics of interactive systems [Mark 81, Wilczynski 81]. Consul's model will be used by and extended by CUE to provide the independent semantic description that allows the system to relate the data and functions of different services.

CUE's user interface will focus on techniques which facilitate inter-service sharing, promote a consistent user view of the system, and guide the user through interactions. A number of existing techniques and extensions to them will influence the design of the user interface. Multi-windowed full-screen editors are a proven technique for coordinating and sharing information between files. It is a simple extension to use this technique to facilitate the sharing of information between services. A functional style of computing [Backus 82] will be used to encourage the combination of functions from different services. The benefits of an omni-present editor were demonstrated in the Sigma message system [Stotz, *et al.* 82]. The extension of this approach to a multi-service environment will encourage a consistent user view by providing a uniform mechanism for the examination and modification of data. Extensive use of forms and menus will be used to help the user specify his task. Help will be an integral part of the omni-present editor to assist the user in the understanding of functions and data structures. In addition, the CUE user will have access to Consul's natural language help and explanation facility.

Ensuring the appropriate functional grain size for each service requires a combination of checking with respect to the semantic model and with respect to the other functions in the system. CUE will use the model to ensure that the size of each new function makes *semantic* sense (i.e., that the function works on the right kind of objects--e.g., that a function that purports to be a "send" puts message data structures into mailbox data structures). It will use its knowledge of the other functions in the system to ensure that each new function makes *structural* sense (e.g., that there are other functions for accessing the mailbox data structure and for composing the messages).

CUE's user interface is designed to interact with a system (Consul) that provides natural request handling and generation of specific help for the user's problems. The CUE user will be able to type sentences such as, "Allow anyone to read this file." or "How can I recover from this error?" CUE will notice that these are not command requests, realize that they might be natural language requests and pass them to Consul. Consul will be be able to process sentences such as these only if it has a model of the current environment. CUE must supply Consul with enough information to build this model. The CUE and Consul projects must together determine how much of the state should be modeled in Consul and what mechanism Consul will use to reference or access information that is not explicitly modeled. The projects must also determine an appropriate formalism for efficient communication at two levels: knowledge base operations and procedure invocation (service procedures in CUE and inference procedures in Consul). From the operating system viewpoint, these are both presumably interprocess communications that will have to be covered by the appropriate protocols (see [Rashid 80]).

3. The CUE System

The CUE system (see Figures 1 and 4) is a computing environment for building and using services. Although both of these activities use the same set of internal mechanisms they give rise to two distinct (external) views of the CUE system: the service builder's and the end user's. For the service builder, CUE is an active programming environment that imposes certain requirements on the way he writes his functions. He is made aware of these requirements via an interactive dialogue with the system. When the requirements are fulfilled, CUE accepts the function, meaning that it becomes a part of its uniform service environment. For the user, CUE is an editor-based, command-oriented interactive system with two important features: (1) it allows him to think only in terms of the commands necessary to perform his task--he does not have to worry about individual service subsystems and their idiosyncratic user interfaces; and (2) it gives him access to knowledge-based help and inference facilities when necessary to aid his interactions.

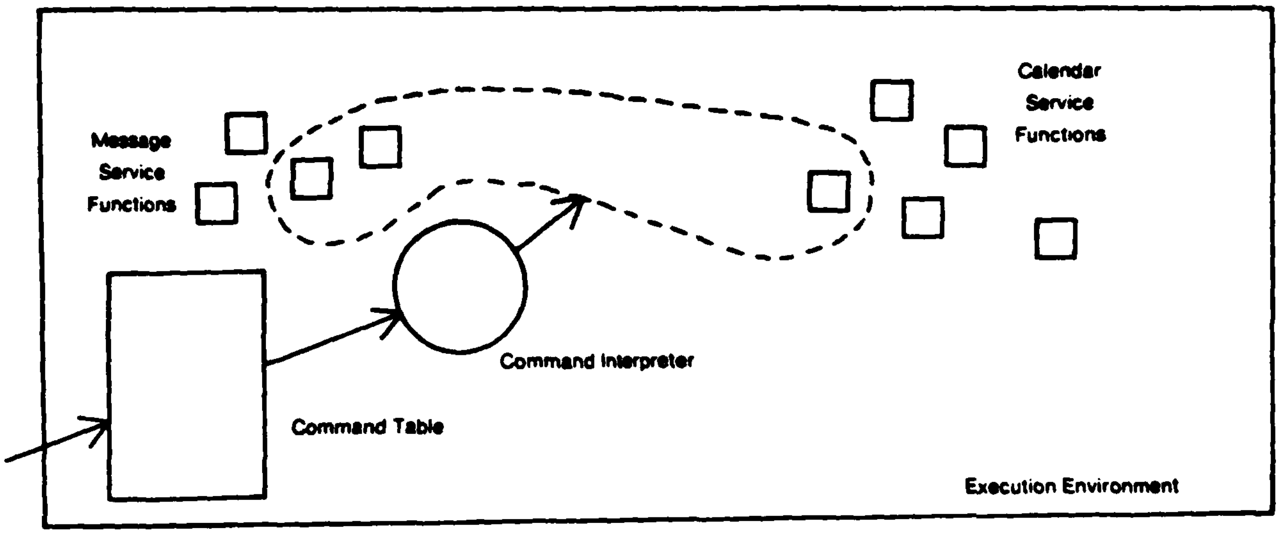


Figure 1: CUE

3.1 The Service Builder's View of CUE

The guiding philosophy of the CUE service building environment is that integrated systems will never result from relying on the "good programming practices" or good will of the individual service builder. The possible interactions between functions and data in a large system are just too complex to expect the service builder to anticipate them and incorporate them into his service. Instead, the system must keep track of these external demands on his service definition and make him aware of them as each function is defined. In this way the service builder can be confident that once CUE has accepted his function, it will be appropriately integrated into the rest of the environment. The semantic models of services that CUE constructs also serve as an additional form of documentation that should prove to be a valuable aid in software maintenance.

3.1.1 Adding a New Service Element

The fundamental service building activity in CUE is the definition of a single new service element (data structure or function) in terms of the process script language. This is a program declaration and configuration formalism that has been developed (to a degree) for use in the Consul system [Lingard 81]. The formalism must be extended and implemented as an efficient computation environment to fulfill the needs of CUE. Process scripts consist of a descriptive part, used to declare data structures and function invocation specifications, and a procedural part, used to configure these data structures and functions into task-level routines (much like the UNIX "shell" language). The functions themselves can be written in any programming language, as far as CUE is concerned.

Entry of a process script definition triggers an acquisition dialogue, CUE's mechanism for relating the new element to all other relevant elements in the system. This integration is accomplished by examination of the knowledge base, the data structures declared in previously entered process scripts, and the existing command table.

The knowledge base consists of a large structure of KL-ONE "concepts" [Brachman 78], semantic representations of the elements of the interactive systems world (a message, the delete operation, a sending event, etc.). CUE's acquisition process attempts to relate each incoming service element to one or more of these concepts, basically by checking the concept structure component by component against the process script definition. Once the element has been associated with a concept structure, CUE has it (semantically) pegged: it knows what it is supposed to be used for or what it is supposed to do. This enables CUE to isolate the other data structures and functions that are semantically related to the new element. The new element is then checked with respect to these related elements to make sure that it is consistent with the overall system framework.

CUE must next make sure that the new element is structurally consistent with the semantically relevant elements. That is, given the existing process script declarations, command table entries, and common data representations, does the new element fit into the CUE execution environment the way the model thinks it should (is it accessible or executable by everything with which it is semantically related)? In other words, CUE must confirm that the element can be *used* in an integrated way (e.g., that the data structure representing the attendee of a meeting can really be used to form the data structure that represents the addressee in a call to *SendMessage*).

Thus, when CUE knows from its knowledge base association what information the data is supposed to contain, it must check to ensure that the representation of this information is compatible with the representations expected by the functions that can use this information. This means making certain that the new element has a common data definition or that the system contains appropriate coercion routines to transform it into what the relevant functions expect. If CUE does not have appropriate coercion functions, it can request the service builder to provide them, construct them itself, or build appropriate tables to support coercion.

For example, suppose that two services use different packed representations of the date. When CUE notices this (during acquisition) it can construct coercion functions which:

- use the access functions of the source data type to isolate its fields,
- use the unpack operators (if necessary) to convert these to primitive data types, and finally,
- use the constructor of the target data type to build the result.

CUE is guaranteed to know the access, unpack, and constructor operators for the data types because of CUE's insistence that all data be defined using abstract data types.

The construction of a data base to support coercion can be triggered when CUE notices that the service builders are choosing to concentrate on different attributes of a common concept rather than choosing different representations of the same attributes of a concept. For example, the service builder for a mail system may concentrate on the attributes of a person which allow electronic mail to be delivered whereas the service builder for a calendar system may be concerned only with the names of the people who are to be in attendance. The most straightforward approach CUE can adopt is to build a flat data base with fields that represent all of the possible attributes of a concept and association lists that describe the relationships between the fields of the data base and the fields of the various data types that can be constructed from the information in the data base.

CUE must also check that all the necessary functions are available to support a data type. This involves two levels of checking--one at the semantic level (e.g., if a data type represents messages then customary mail operations should be defined for it) and the other at the data representation level (e.g., if a data type is defined to be a structure, then CUE must know how to access components of the structure and how to construct the structure from its components).

Similarly, when CUE knows what a function is supposed to do, it must check to see that the function's invocation specification is compatible with what is expected by other process scripts, and that its command table entry is consistent with other relevant commands (i.e., the argument structure, undo behavior, etc. of the new command must be consistent with commands of "similar function"). Thus, the function for sending a message must not only be callable by other functions (e.g., one that sends out automatic reminders), but also be invocable by a user command that is consistent with commands for other sending functions (e.g.,forwarding).

CUE's behavior when a new program element is associated with a knowledge base concept that already has associated program elements depends on whether the new element is a data structure or a function. If a concept is associated with multiple data structures, CUE believes they all contain the same information, and will insist on knowing how to transform one into another. If a concept is associated with more than one function, CUE will always come back to the service builder, since there is usually a problem: lack of understanding on CUE's part, insufficient specificity on the service-builder's part, or redundancy--someone else has already defined the function.

3.1.2 Scenario 1: Adding the *Remind* Function

The service builder's view of CUE will be illustrated by a hypothetical scenario of CUE processing. Let us assume that a service-builder wishes to add a function that users can call to remind themselves of upcoming meetings. The idea is to set up a demon that will send a message at some given time before a meeting occurs (if the meeting is no longer on the schedule at the appointed time, the demon does nothing). Note that this function involves inter-service functionality, since it requires the interaction of both electronic mail and calendar functions.

The *SendMessage*, *CheckSchedule*, and *Mailer* functions are already in the system. *SendMessage* takes as arguments an addressee list and a message. The addressee list must consist of users who have mailboxes. *CheckSchedule* determines whether the meeting it is given as argument is still on the schedule. A *meeting* is a data structure with fields *time*, *title*, *subject*, *chairman*, and *attendees*; an attendee is simply a user. The *Mailer* function sends any messages that have been queued.

The process script definition of the *Remind* function is shown in Figure 2 (only a subset of the process script descriptors are shown). *Remind* takes as arguments the user who is to receive the reminder, the length of time before the meeting that the reminder is to be sent, and the meeting the reminder refers to. The function binds a message data structure, sets the subject of the message to some text based on the title of the meeting, and sets up the demon that is going to send the reminder message. The demon's action is determined by another function (described below); its action condition is the scheduled meeting time minus the time interval given as argument.

When the service builder enters this process script, an acquisition dialogue is triggered. The acquisition mechanism must be given a starting point, so it asks the service builder what kind of function *Remind* is, giving him a choice of ten or so basic function types (move, delete, display, etc.).

The service builder decides that remind is a kind of demon-setting function. and the modelling process begins. This is basically a step-by-step question and answer session (too detailed to present here) in which the acquisition mechanism attempts to associate each feature of its model of demon-setting functions with some feature of the *Remind* function.

CUE is checking to see whether this new element is of an appropriate grain size to fit in with the rest of the system. and whether the given data structure and function invocation specifications make sense semantically. In this case. *Remind* fits well into the pattern of demon-setting functions: its arguments are well defined data structures. and it does what it is supposed to--sets up a demon with an action and a condition. CUE notes that the action must be checked for semantic consistency when it is defined. and goes on to check the demon condition. Here the question is whether the expression (m.time - tbm) is a valid demon condition. CUE uses the semantic model to determine that this expression denotes a time. and checks to see whether its model of demons allows a time specification as a valid condition--which it does.

```
ProcessScript  Remind:
  Input          u:User. tbm:Time. m:Meeting:
  Output         none:
  SideEffects    none:
  Undo           none;

    Local msg:Message:

    msg.subject <- Concatenate(m.title.
                         'MEETING AT '.
                         m.time):
    Call SetDemon(ReminderDemonAction(u. m. msg. tbm).
              (m.time - tbm)):
end:
```

Figure 2: Process Script Definition of "Remind"

CUE next checks to see whether the *Remind* process script makes sense structurally by examining all of its declared data structures and invocation specifications with respect to the other declarations in the system. This may also require reference to the semantic model to access the entire body of structural knowledge that is known about a particular *kind* of program element (e.g., to see its various data representations). For example. CUE discovers (by looking at the specification for *Concatenate*) that *m.title* must have a string representation. Let us assume that the title field of the meeting data structure does not have a string representation. Nonetheless, by noting that the title is a kind of text. CUE finds that it can be represented as a string. It will also discover that meetings do not have to have titles and warn the service builder. The service builder may choose to rewrite the script to use the meeting subject if there is no title. CUE would then note that meetings do not have to have subjects either. in which case the service builder might resort to the meeting chairman. and then finally a null string. Similarly. CUE will ensure that the different time representations can be made consistent to compute a demon condition that is acceptable to the demon firing mechanism.

Finally, CUE will add a command invocation for *Remind* in the command table. making sure that it is consistent with its other demon-setting commands.

The same kind of acquisition activity occurs for the definition of the reminder demon action itself (see Figure 3). Here the function is recognized semantically as a kind of send operation, meaning (among other things) that the argument *msg* must really be a message and that everyone on the addressee list--just *u* in this case--must have a mailbox. The fact that *msg* is a message is guaranteed (semantically) by its explicit declaration as a *Message*. However, CUE cannot determine whether the user *u* necessarily has a mailbox, and warns the service builder of that fact. The service builder will have to change his specification of *u* (in the original *Remind* process script) to be restricted to addressees, i.e., users with mailboxes. This change propagates back to the command table entry for *Remind*, enabling CUE to check the validity of the remindee at the time the end user invokes the remind command.

```
ProcessScript  ReminderDemonAction;
 Input          u:User, m:Meeting, msg:Message, tbm:Time;
 Output         none;
 SideEffects    none;
 Undo           none;

  if CheckSchedule(m)
   then do; SendMessage((u), msg);
          if tbm < 60 minutes
             then Call Mailer;
       end;
   else;
end;
```

Figure 3: Process Script Definition of "ReminderDemonAction"

The structural checking of *ReminderDemonAction* is also similar to that for *Remind*. This time CUE makes sure that the given string is a valid message subject, that the time parameter can be converted for comparison with "60 minutes", etc.

Since *ReminderDemonAction* is not an end user level function, no command table entry is made for it.

3.2 The User's View of CUE

The user sees CUE as a uniform interactive environment for performing his tasks. When he wants the system to do something, he gives the appropriate series of commands. Alternatively, he can give the system a natural language request to do his task (in which case he can expect his request to take considerably more processing time). If he needs information on some aspect of the system or makes an error, he expects a response whose character (and processing time) are decided by his choice of an "explanation level" setting: he can arrange for rapid-response canned help and error messages or longer-response natural language explanations that are directed to his specific problem. At no time does the user have to think in terms of which particular services are performing the various aspects of his task.

This kind of interaction will be achieved in CUE by (1) insisting that services be constructed according to the methodology described in the previous section, (2) providing a uniform command interpreter with a help facility, and (3) having the mechanism to pass natural language requests and explanation situations to Consul. CUE's user interface philosophy follows the escalation principle: "send problematic inputs on to more and more sophisticated mechanisms". Thus, a user request that corresponds to one of the formal commands in CUE's command table would be handled in much the same way as existing systems process commands. A request of command form that contains discrepant parameter specifications would be checked with respect to knowledge CUE has about data types. This knowledge, resulting from acquisition with respect to CUE's semantic models, allows interpretation and (if necessary) automatic coercion. Finally, a user request that does not correspond to any command would be sent on to the Consul inference mechanisms (perhaps operating on a more powerful processor) for parsing and reformulation.

CUE also allows the user to block escalation of command and explanation processing and confine all processing to his local CUE machine. In this mode, CUE's input processing results in behavior similar to that of current interactive systems except for the absence of service boundaries. That is, the user can freely intermix commands for action in different services, and can refer to the objects of any service domain in his commands.

CUE can thus be operated in a "communication" mode (see "CUE 2" in Figure 4) when the user wishes to have knowledge-based services available (and when a Consul machine is available), or in a "stand-alone" mode (see "CUE 1" in Figure 4) when the user wishes to have minimum response time (or to continue operation when no Consul machine is available). The CUE system will therefore be able to function either as a part of a distributed cooperative interface environment, or as a complete and separate personal machine environment.

This methodology makes CUE the ideal vehicle for making "advanced" knowledge-based technology available to a wide audience.

3.2.1 Scenario 2: Performing the User's Task

The following hypothetical scenario illustrates CUE activity for the user task:

sending a message to everyone in my 3:30 meeting to tell them I can't make it

Let us see how the user might specify this task in command form. Suppose that the user has just discovered an urgent task to perform and that it will take the rest of the day. Checking the schedule to see if there are any important appointments, the user sees that the only one of interest is a 3:30 meeting. A full display of the information about this meeting is brought up in a window on the screen. The user then invokes the "Send" function, which causes a Send-Message-form to be displayed with a number of fields including a "Text:" field and a "To:" field. Defaults for other fields such as the "From:" and "CC:" fields are filled in on the form. The user types a message into the text field and then moves the cursor to the display of the list of attendees of the meeting, uses function keys to create a copy of this list, moves the cursor to the "To:" field of the Send-Message-form and finally, uses the function keys to indicate that this is where to place the copy.

CUE must check to see whether the "To:" argument is a valid addressee list; it is not. However, CUE does know that the argument is a valid attendee list, and checks to see whether it has an explicit transformation for converting attendee lists into addressee lists. It does not. Still, CUE knows that both of these data structures are lists, and that if it can convert each element of the attendee list into an addressee, a valid addressee list will result. Accordingly, it checks to see whether there is a transformation that takes attendees into addressees. CUE looks at what it knows about the data types and finds that it has a data base which contains information about people known to CUE. CUE uses the names of the attendees of the meeting as keys to access records from this data base. It then extracts the appropriate fields from the records and constructs an addressee data structure for each attendee. A list is constructed from these addressees and put into the "To:" field of the message. The message is now appropriate for transmission.

This is not a special case that CUE could resolve because the data base just happened to exist. The data base exists because the service builders of the message and calendar services went through the acquisition process and informed CUE that attendees and addressees were both people. Since there were two data structures associated with the same concept, CUE informed the service builders of the situation and constructed a data base of information about people. Presumably other services also have data structures that represent people: the data base must accommodate them all. The approach used is limited only by the generality of the model that exists before the service builder begins the acquisition process. The goal of the CUE project is to provide enough model to support the variety of general services that are normally found in an interactive system (e.g., mail, document preparation, calendars, etc). If specialized applications (e.g., software maintenance, signal processing, VLSI design, personnel files, etc) are to be supported, then the problem domains of these applications must be modeled.

CUE could decompose and construct the data structures involved in this example because it is aware of which fields of the records to access (knowledge base associations built at acquisition time prescribed them) and because the abstract data type formalism requires the operators of a data type to be explicitly identified and accessible.

Note that although the acquisition process requires sophisticated processing to construct and maintain elaborate knowledge representation models, run-time support for CUE requires much more modest resources. This points out one of the real benefits of CUE: it makes use of sophisticated (and expensive) artificial intelligence knowledge representation only when necessary--there is little or no overhead for routine interactions. During run-time the only information that CUE needs from the knowledge base is the relations between concepts. This information can be distilled from the knowledge base and restructured into a more efficient tabular representation. But because the semantic models were used to integrate the elements of the system, it is guaranteed to behave in a consistent manner with increased functionality (due mainly to automatic data coercion). As an added feature the user has access to services through Consul's more sophisticated, knowledge-based user interface.

3.3 System Configuration

Given the computational needs and probable machine resources for the various parts of the system, we expect the CUE system to appear as shown in Figure 4. The knowledge base and the Consul system will all exist on a central shared machine. All actual service programs, the omnipresent editor, the local command interpreter, and the service computation environment will reside on single user machines that have the capability (but not the requirement) of communication with the Consul machine.

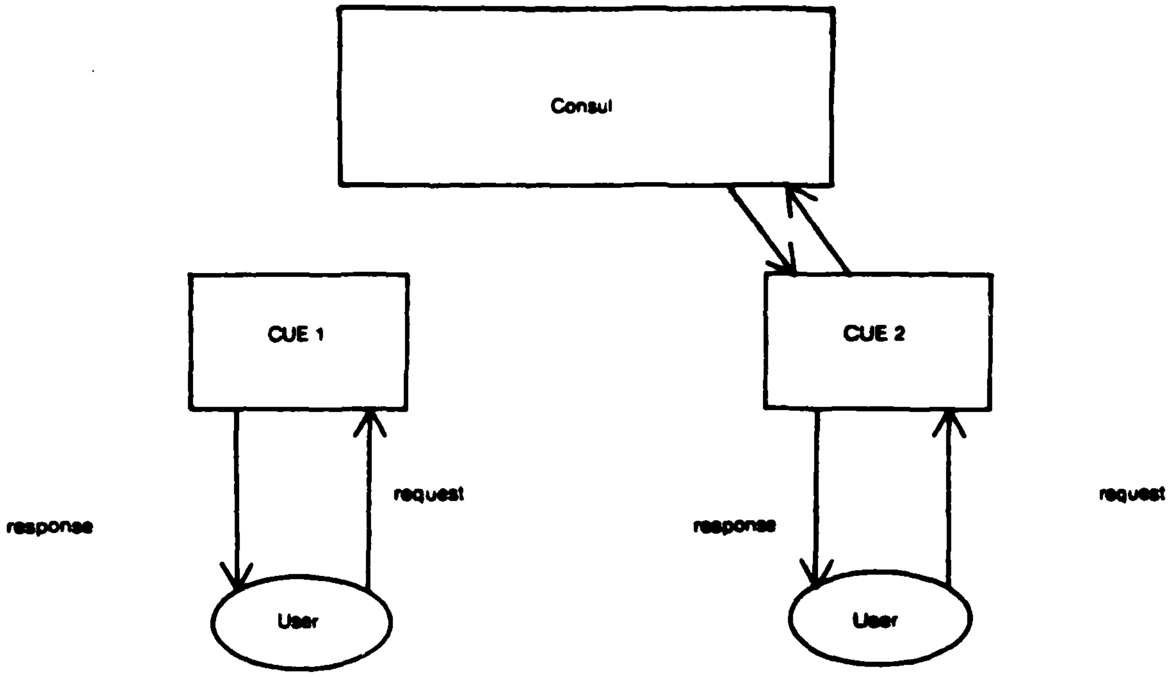


Figure 4: CUE Machine Configuration

3.4 Summary

In summary, the CUE Project will:

[1] provide an environment that allows cooperation among several services to solve a common task;

[2] produce a self-contained, command-oriented, editor-based personal working environment;

[3] allow interaction with Consul in order to provide a knowledge-based user interface;

[4] implement useful services in the integrated framework.

References

[Backus 82] John Backus. "Function-level Computing." *IEEE Spectrum* 19. (8). August 1982. 22-28.

[Brachman 78] Ronald Brachman. *A Structural Paradigm for Representing Knowledge*. Bolt. Beranek. and Newman. Inc.. Technical Report. 1978.

[Dahl 72] O.J.Dahl, E.W.Dijkstra and C.A.R.Hoare. *A.P.I.C.Studies in Data Processing.* Volume 8: *Structured Programming*. Academic Press. 1972.

[IPAD 80] NASA, *IPAD: Integrated Programs For Aerospace-Vehicle Design*, NASA, 1980.

[Lingard 81] Robert Lingard, "A Software Methodology for Building Interactive Tools," in *Proceedings of the Fifth International Conference on Software Engineering*. 1981.

[Mark 81] William Mark. "Representation and Inference in the Consul System." in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. IJCAI. 1981.

[Rashid 80] Richard Rashid. *An Inter-Process Communication Facility for UNIX*. Canegie-Mellon University, Technical Report. March 1980.

[Stotz, *et al.* 82] R. Stotz. *et al.*, *SIGMA Final Report*. Information Sciences Institute. Technical Report ISI/RR-82-94 and ISI/RR-82-95. 1982.

[Wilczynski 81] David Wilczynski. "Knowledge Acquisition in the Consul System." in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. IJCAI. 1981.

[Yntema 73] D.B. Yntema. *The Cambridge Project: Computer Methods for Analysis and Modeling of Complex Systems*. Rome Air Development Center. Technical Report. 1973.